Why Amazon Chose TLA$^+$

Chris Newcombe

Amazon, Inc.

Abstract. Since 2011, engineers at Amazon have been using TLA$^+$ to help solve difficult design problems in critical systems. This paper describes the reasons why we chose TLA$^+$ instead of other methods, and areas in which we would welcome further progress.

1 Introduction

Why Amazon is using formal methods. Amazon builds many sophisticated distributed systems that store and process data on behalf of our customers. In order to safeguard that data we rely on the correctness of an ever-growing set of algorithms for replication, consistency, concurrency-control, fault tolerance, auto-scaling, and other coordination activities. Achieving correctness in these areas is a major engineering challenge as these algorithms interact in complex ways in order to achieve high-availability on cost-efficient infrastructure whilst also coping with relentless rapid business-growth[1]. We adopted formal methods to help solve this problem.

Usage so far. As of February 2014, we have used TLA$^+$ on 10 large complex real-world systems. In every case TLA$^+$ has added significant value, either preventing subtle serious bugs from reaching production, or giving us enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness. Executive management are now proactively encouraging teams to write TLA$^+$ specs for new features and other significant design changes. In annual planning, managers are now allocating engineering time to use TLA$^+$.

We lack space here to explain the TLA$^+$ language or show examples of our specifications. We refer readers to [18] for a tutorial, and [26] for an example of a TLA$^+$ specification from industry that is similar in size and complexity to some of the larger specifications at Amazon.

What we wanted in a formal method. Our requirements may be roughly grouped as follows. These are not orthogonal dimensions, as business requirements and engineering tradeoffs are rarely crisp. However, these do represent the most important characteristics required for a method to be successful in our industry segment.

1. *Handle very large, complex or subtle problems.* Our main challenge is complexity in concurrent and distributed systems. As shown in Fig. 1, we often need to verify the interactions between algorithms, not just individual algorithms in isolation.

[1] As an example of such growth; in 2006 we launched S3, our Simple Storage Service. In the six years after launch, S3 grew to store 1 trillion objects [6]. Less than one year later it had grown to 2 trillion objects, and was regularly handling 1.1 million requests per second [7].

System	**Components**	**Line count**	**Benefit**
S3	Fault-tolerant low-level network algorithm	804 PlusCal	Found 2 design bugs. Found further design bugs in proposed optimizations.
	Background redistribution of data	645 PlusCal	Found 1 design bug, and found a bug in the first proposed fix.
DynamoDB	Replication and group-membership systems (which tightly interact)	939 TLA$^+$	Found 3 design bugs, some requiring traces of 35 steps.
EBS	Volume management	102 PlusCal	Found 3 design bugs.
EC2	Change to fault-tolerant replication, including incremental deployment to existing system, with zero downtime	250 TLA$^+$ 460 TLA$^+$ 200 TLA$^+$	Found 1 design bug.
Internal distributed lock manager	Lock-free data structure	223 PlusCal	Improved confidence. Failed to find a liveness bug as we did not check liveness.
	Fault tolerant replication and reconfiguration algorithm	318 TLA$^+$	Found 1 design bug. Verified an aggressive optimization.

Fig. 1. Examples of applying TLA$^+$ to some of our more complex systems

Also, many published algorithms make simplifying assumptions about the operating environment (e.g. fail-stop processes, sequentially consistent memory model) that are not true for real systems, so we often need to modify algorithms to cope with the weaker properties of a more challenging environment. For these reasons we need expressive languages and powerful tools that are equipped to handle high complexity in these problem domains.

2. *Minimize cognitive burden.* Engineers already have their hands full with the complexity of the problem they are trying to solve. To help them rather than hinder, a new engineering method must be relatively easy to learn and easy to apply. We need a method that avoids esoteric concepts, and that has clean simple syntax and semantics. We also need tools that are easy to use. In addition, a method intended for specification and verification of designs must be easy to remember. Engineers might use a design-level method for a few weeks at the start of a project, and then not use it for many months while they implement, test and launch the system. During the long implementation phase, engineers will likely forget any subtle details of a design-level method, which would then cause frustration during the next design phase. (We suspect that verification researchers experience a very different usage pattern of tools, in which this problem might not be apparent.)
3. *High return on investment.* We would like a single method that is effective for the wide-range of problems that we face. We need a method that quickly gives useful results, with minimal training and reasonable effort. Ideally we want a method that also improves time-to-market in addition to ensuring correctness.

Comparison of methods. We preferred candidate methods that had already been shown to work on problems in industry, that seemed relatively easy to learn, and that we could apply to real problems while we were learning. We were less concerned about "verification philosophy" such as process algebra vs. state machines; we would have used any method that worked well for our problems.

We evaluated Alloy and TLA$^+$ by trying them on real-world problems [29]. We did a smaller evaluation of Microsoft VCC. We read about Promela/Spin [13], Event-B, B, Z, Coq, and PVS, but did not try them, as we halted the investigation when we realized that TLA$^+$ solved our problem.

2 Handle Very Large, Complex or Subtle Problems

2.1 Works on Real Problems in Industry

TLA$^+$ has been used successfully on many projects in industry: complex cache-coherency protocols at Compaq [25] and Intel [8], the Paxos consensus algorithm [21], the Farsite distributed file system [9], the Pastry distributed key-value store [28], and others. Alloy has been used successfully to find design flaws in the Chord ring membership protocol [32] and to verify an improved version [33]. Microsoft VCC has been used successfully to verify the Microsoft Hypervisor [27].

Event-B [5] and B [1] have been used in industry, although most of the applications appear to be control systems, which is an interesting area but not our focus. We found some evidence [12] that Z has been used to specify systems in industry, but we did not find evidence that Z has been used to verify large systems. PVS has been used in industry but most of the applications appear to be low-level hardware systems [30]. We did not find any relevant examples of Coq being used in industry.

2.2 Express Rich Dynamic Structures

To effectively handle complex systems, a specification language must be able to capture rich concepts without tedious workarounds. For example, when modelling replication algorithms we need to be able to specify dynamic sequences of many types of nested records. We found that TLA$^+$ can do this simply and directly. In contrast, all structures in Alloy are represented as relations over uninterpreted atoms, so modelling nested structures requires adding intermediate layers of identifiers. This limitation deters us from specifying some distributed algorithms in Alloy. In addition, Alloy does not have a built-in notion of time, so mutation must be modelled explicitly, by adding a timestamp column to each relation and implementing any necessary dynamic structures such as sequences. This has occasionally been a distraction.

Alloy's limited expressiveness is an intentional trade-off, chosen to allow analysis of very abstract implicit specifications[2]. In contrast, TLA$^+$ was designed for clarity

[2] The book on Alloy [15, p. 302] says, "... the language was stripped down to the bare essentials of structural modelling, and developed hand-in-hand with its analysis. Any feature that would thwart analysis was excluded", and [15, p. 41], "The restriction to flat relations makes the logic more tractable for analysis."

of expression and reasoning, initially without regard to the consequences for analysis tools[3]. In practice, we have found that the model checker for TLA$^+$ is usually capable of model-checking instances of real-world concurrent and distributed algorithms at least as large as those checkable by the Alloy Analyzer. However, the Alloy Analyzer can perform important types of inductive analysis (e.g. [33, p. 9]) that the currently available tools for TLA$^+$ cannot handle. We have not yet tried inductive analysis of real-world algorithms, but are interested in doing so for reasons discussed later in this paper.

2.3 Easily Adjust the Level of Abstraction

To verify complex designs we need to be able to add or remove detail quickly, with reasonably small localized edits to the specification. TLA$^+$ allows this by supporting arbitrarily complicated data structures in conjunction with the ability to define powerful custom operators[4] that can abstract away the details of those structures. We found that Alloy is significantly less flexible in this area as it does not support recursive operators or functions [15, pp. 74,280], or higher-order functions. Alloy does compensate to some extent by having several built-in operators that TLA$^+$ lacks, such as transitive closure and relational join, but we found it easy to define these operators in TLA$^+$. VCC allows the user to write "ghost code" in a superset of the C programming language. This is an extremely powerful feature, but the result is usually significantly more verbose than when using TLA$^+$ or Alloy. We have not investigated the extensibility or abstraction features of other methods.

2.4 Verification Tools That Can Handle Complex Concurrent and Distributed Systems

The TLA$^+$ model-checker works by explicitly enumerating reachable states. It can handle large state-spaces at reasonable throughput. We have checked spaces of up to 31 billion states[5], and have seen sustained throughput of more than 3 million states per minute when checking complex specifications. The model checker can efficiently use multiple cores, can spool state to disk to check state-spaces larger than physical memory, and supports a distributed mode to take good advantage of additional processors, memory and disk. Distributed mode has been very important to us, but we have also obtained good results on individual 32-core machines with local SSD storage.

At the time of our evaluation, Alloy used off the shelf SAT solvers that were limited to using a single core and some fraction of the physical RAM of a single machine. For moderate finite scopes the Alloy Analyzer is extremely fast; much faster than the TLA$^+$

[3] The introductory paper on TLA$^+$ [19] says, "We are motivated not by an abstract ideal of elegance, but by the practical problem of reasoning about real algorithms … we want to make reasoning as simple as possible by making the underlying formalism simple." The designer of TLA$^+$ has expressed surprise that the language could be model-checked at all [16]. The first model-checker for TLA$^+$ did not appear until approximately 5 years after the language was introduced.

[4] TLA$^+$ supports defining second-order operators [20, p. 318] and recursive functions and operators with a few restrictions.

[5] Taking approximately 5 weeks on a single EC2 instance with 16 virtual CPUs, 60 GB RAM and 2 TB of local SSD storage.

model checker. However, we found that the SAT solvers often crash or hang when asked to solve the size of finite model that is necessary for achieving reasonable confidence in a more complex concurrent or distributed system. When we were preparing this paper, Daniel Jackson told us [14] that Alloy was not intended or designed for model checking such algorithms. Jackson says, "Alloy is a good choice for direct analysis if there is some implicitness in the specification, e.g. an arbitrary topology, or steps that can make very unconstrained changes. If not, you may get Alloy to be competitive with model checkers, but only on short traces." However we feel that Jackson is being overly modest; we found Alloy to be very useful for understanding and debugging the significant subset of concurrent algorithms that have relatively few variables, shallow nesting of values, and for which most interesting traces are shorter than perhaps 12 to 14 steps. In such cases, Alloy can check sophisticated safety properties sufficiently well to find subtle bugs and give high confidence—as Pamela Zave found when using Alloy to analyze the Chord ring membership protocol [32,33].

VCC uses the Z3 SMT solver, which is also limited to a single machine and physical memory (we are not sure if it uses multiple cores). We have not yet tried the ProB model checker for B or Event B, but we believe that it is also limited to using a single core and physical RAM on a single machine.

Another limitation of Alloy is that the SAT-based analysis can only handle a small finite set of contiguous integer values. We suspect that this might prevent Alloy from effectively analyzing industrial systems that rely on properties of modulo arithmetic, e.g. Pastry. TLA$^+$ was able to specify, model-check, and prove Pastry [28].

2.5 Good Tools to Help Users Diagnose Bugs

We use formal methods to help us find and fix very subtle errors in complex designs. When such an error is found, the cause can be difficult for the designer to diagnose. To help with diagnosis we need tools that allow us to comprehend lengthy sequences of complex state changes. This is one of the most significant differences between the formal methods that we tried. The primary output of the Alloy Analyzer tool is diagrams; it displays an execution trace as a graph of labelled nodes. The tool uses relatively sophisticated algorithms to arrange the graph for display, but we still found this output to be incomprehensible for systems with more than a few variables or time-steps. To work around this problem we often found ourselves exporting the execution trace to an XML file and then using a text editor and other tools to explore it. The Alloy Analyzer also has a feature that allows the user to evaluate arbitrary Alloy expressions in the context of the execution trace. We found the evaluator feature to be very useful (although the versions we used contain a bug that prevents copy/paste of the displayed output). The TLA$^+$ Toolbox contains a similar feature called Trace Explorer that can evaluate several arbitrary TLA$^+$ expressions in every state of an execution trace and then display the results alongside each state. We found that this feature often helped us to diagnose subtle design errors. At the time of writing, Trace Explorer still has some quirks: it is somewhat slow and clunky due to having to launch a separate short-lived instance of the model checker on every change, and the IDE sometimes does a poor job of managing screen space, which can cause a tedious amount of clicking, resizing, and scrolling of windows.

2.6 Express and Verify Complex Liveness Properties

While safety properties are more important than liveness properties, we have found that liveness properties are still important. For example, we once had a deadlock bug in a lock-free algorithm that we caught in stress testing after we had failed to find it during model-checking because we did not check liveness. TLA$^+$ has good support for liveness. The language can express fairness assumptions and rich liveness properties using operators from linear time temporal logic. The model checker can check a useful subset of liveness properties, albeit only for very small instances of the system, as the algorithm for checking liveness is restricted to using a single core and physical RAM. However, the current version of the model checker has a significant flaw: if a liveness property is violated then the model checker may report an error trace that is much more complicated than necessary, which makes it harder for the user to diagnose the problem. For deeper verification of liveness, the TLA$^+$ proof system will soon support machine-checked proofs using temporal logic. The Alloy book says that it supports checking liveness[6], but only on short traces[7]. When evaluating Alloy, Zave found [33], "There are no temporal operators in Alloy. Strictly speaking the progress property could be expressed in Alloy using quantification over timestamps, but there is no point in doing so because the Alloy Analyzer could not check it meaningfully ... For all practical purposes, progress properties cannot be asserted in Alloy." VCC can verify local progress properties, e.g. termination of loops and functions, but we don't know if VCC can express fairness or verify global liveness properties.

2.7 Enable Machine-Checked Proof

So far, the benefits that Amazon have gained from formal methods have arisen from writing precise specifications to eliminate ambiguity, and model-checking finite models of those specifications to try to find errors. However, we have already run into the practical limits of model-checking, caused by combinatorial state-explosion. In one case, we found a serious defect that was only revealed in an execution trace comprising 35 steps of a high-level abstraction of a complex system. Finding that defect took several weeks of continuous model-checking time on a cluster of 10 high-end machines, using carefully crafted constraints to bound the state-space. Even when using such constraints the model-checker still has to explore billions of states; we had to request several enhancements to the model checker to get even this far. We doubt that model-checking can go much further than this. Therefore, for our most critical algorithms we wish to also use proofs.

In industry, engineers are extremely skeptical of proofs. Engineers strongly doubt that proofs can scale to the complexity of real-world systems, so any viable proof method would need an effective mechanism to manage that complexity. Also, most

[6] The Alloy book [15, p. 302] says, "[Alloy] assertions can express temporal properties over traces", and [15, p. 179] "we'll take an approach that ... allows [the property 'some leader is eventually elected'] to be checked directly." See also [15, p. 186], "How does the expressiveness of Alloy's trace assertions compare to temporal logics?"

[7] [15, p. 187] "In an Alloy trace analysis, only traces of bounded length are considered, and the bound is generally small."

proofs are so intricate that there is more chance of an error in the proof than an error in the algorithm, so for engineers to have confidence that a proof is correct, we would need machine verification of the proof. However, most systems that we know of for machine-checked proof are designed for proving conventional theorems in mathematics, not correctness of large computing systems.

TLA$^+$ has a proof system that addresses these problems. The TLA$^+$ proof system (TLAPS) uses structured hierarchical proof, which we have found to be an effective method for managing very complex proofs. TLAPS works directly with the original TLA$^+$ specification, which allows users to first eliminate errors using the model checker and then switch to proof if even more confidence is required. TLAPS uses a declarative proof language that emphasizes the basic mathematics of the proof and intentionally abstracts away from the particular languages and tactics of the various back-end proof checkers. This approach allows new proof checkers to be added over time, and existing checkers to evolve, without risk of requiring significant change to the structure or content of the proof. If a checker does happen to change in a way that prevents an existing step from being proved, the author can simply expand that local step by one or more levels.

There are several published examples of TLAPS proofs of significant algorithms [28,22,11]. We have tried TLAPS on small problems and found that it works well. However, we have not yet proved anything useful about a real system. The main barrier is finding inductive invariants of complex or subtle algorithms. We would welcome tools and training in this area. The TLA$^+$ Hyperbook [18] contains some examples but they are relatively simple. We intend to investigate using Alloy to help debug more complex inductive invariants, as Alloy was expressly designed for inductive analysis.

While preparing this paper we learned that several proof systems exist for Alloy. We have not yet looked at any of these, but may do so in future.

VCC works entirely via proof but also provides many of the benefits of a model checker; if a proof fails then VCC reports a concrete counter-example (a value for each variable) that helps the engineer understand how to change the code or invariants to allow the proof to succeed. This is an immensely valuable feature that we would strongly welcome in any proof system. However, VCC is based on a low level programming language, so we do not know if it is a practical tool for proofs of high-level distributed systems.

3 Minimize Cognitive Burden

3.1 Avoid Esoteric Concepts

Methods such as Coq and PVS involve very complicated type-systems and proof languages. We found these concepts to be difficult to learn. TLA$^+$ is untyped, and through using it we have become convinced that a type system is not necessary for a specification method, and might actually be a burden as it could constrain expressiveness. In a TLA$^+$ specification, “type safety” is just another (comparatively shallow) property that the system must satisfy, which we verify along with other desired properties. Alloy has a very simple type system that can only prevent a few trivial kinds of error in a specification. In fact, the Alloy book says [15, p. 119], “Alloy can be regarded as an untyped language". VCC has a simple type system, but adds some unfamiliar concepts such as "ghost claims" for proving safe access to objects in memory. We did not use these features in our evaluation of VCC, so we don't know if they complicate or simplify verification.

3.2 Use Conventional Terminology

TLA$^+$ largely consists of standard discrete math plus a subset of linear temporal logic. Alloy is also based on discrete math but departs from the standard terminology in ways that we initially found confusing. In Alloy, a set is represented as a relation of arity one, and a scalar is represented as a singleton set. This encoding is unfamiliar, and has occasionally been a distraction. When reasoning informally about a formula in Alloy, we find ourselves looking for context about the arity and cardinality of the various values involved in order to deduce the semantics.

3.3 Use Just a Few Simple Language Constructs

TLA$^+$ uses a small set of constructs from set theory and predicate logic, plus a new but straightforward notation for defining functions [17]. TLA$^+$ also includes several operators from linear temporal logic that have semantics that are unfamiliar to most engineers in industry. However, TLA$^+$ was intentionally designed to minimize use of temporal operators, and in practice we've found that we rarely use them. Alloy has a single underlying logic based on the first order relational calculus of Tarski [14], so the foundations of Alloy are at least as simple as those of TLA$^+$. However, the documentation for Alloy describes the logic in terms of three different styles: conventional first-order predicate calculus, "navigation expressions" (including operators for relational join, transpose, and transitive closure), and relational calculus (including operators for intersection and union of relations). All three styles of logic can be combined in the same formula. We found that learning and using this combined logic took a bit more effort than using the non-temporal subset of TLA$^+$, perhaps due to the "paradox of choice" [31].

3.4 Avoid Ambiguous Syntax

TLA$^+$ has a couple of confusing syntactic constructs [20, p. 289], but they rarely arise in practice. The one example that the author has encountered, perhaps twice in four years of use, is an expression of the form $\{x \in S : y \in T\}$. This expression is syntactically ambiguous in the grammar for TLA$^+$ set constructors. The expression is given a useful meaning via an ad hoc rule, but the visual ambiguity still causes the reader a moment of hesitation.

Alloy has a significant amount of syntactic overloading: the "_[_]" operator is used for both applying predicates and functions to arguments [15, p. 123] and for a variation of relational join [15, p. 61]; the "_._" operator is used for both relational join [15, p. 57] and the "syntactic pun" of receiver syntax [15, pp. 124,126]; some formulas ('signature facts') look like normal assertions but contain implicit quantification and implicit relational joins [15, pp. 99,122]; some keywords are reused for different concepts, e.g. for quantifiers and multiplicities [15, p. 72]. Each of these language features is reasonable in isolation, but collectively we found that these details can be a burden for the user to remember. TLA$^+$ has relatively few instances of syntax overloading. For instance, the CASE value-expression uses the symbol $\Box$ as a delimiter but elsewhere that symbol is used for the temporal logic operator that means "henceforth". However, we rarely use CASE value-expressions in our specifications, so this is a minor issue compared to the overloading of the primary operators in Alloy. In the TLA$^+$ proof system, the CASE keyword is overloaded with a meaning entirely different to its use in specifications. However, the context of proof vs. specification is always clear so we have not found this overloading to be a problem.

In Alloy, variable names (fields) can be overloaded by being defined in multiple signatures [15, pp. 119,267], and function and predicate names can be overloaded by the types of their arguments. Disambiguation is done via the static type system. We try to avoid using these features as they can make specifications harder to understand. TLA$^+$ forbids redefinition of symbols that are in scope, even if the resulting expression would be unambiguous [20, p. 32].

3.5 Simple Semantics

TLA$^+$ has the clean semantics of ordinary mathematics. There are a few subtle corner cases: e.g. the ENABLED operator may interact in surprising ways with module instantiation, because module instantiation is based on mathematical substitution and substitution does not distribute over ENABLED. However, such cases have not been a problem for us (we suspect that the main impact might be on formal proof by refinement, which we have not yet attempted).

The Alloy language has some surprising semantics in the core language, e.g. for integers. The Alloy book states [15, p. 82,290] that the standard theorem of integer arithmetic,

$$S =< T \text{ and } T =< S \text{ implies } S = T$$

is not valid in Alloy. The book admits that this is "somewhat disturbing". In Alloy, when S and T each represent a set of integers, the integer comparison operator performs an implicit summation but the equality operator does not. (The equality operator considers two sets of integers to be equal if and only if they contain the same elements.) Alloy has a feature called 'signature facts' that we have found to be a source of confusion. The Alloy book [15, p. 122] says, "The implicit quantification in signature facts can have unexpected consequences", and then gives an example that is described as "perhaps the most egregiously baffling". Fortunately 'signature facts' are an optional feature that can easily be avoided, once the user has learned to do so.

VCC has semantics that are an extension of the C programming language, which is not exactly simple, but is at least familiar to engineers. At the time of our evaluation, VCC had several constructs with unclear semantics (e.g. the difference between 'pure ghost' functions and 'logic' functions), as the language was still evolving, and up-to-date documentation and examples were scarce.

3.6 Avoid Distorting the Language in Order to Be More Accessible

Most engineers find formal methods unfamiliar and odd at first. It helps tremendously if a language has some facilities to help bridge the gap from how engineers think (in terms of programming language concepts and idioms) to declarative mathematical specifications. To that end, Alloy adopts several syntactic conventions from object-oriented programming languages. However, the adoption is superficial, and we have found that the net effect is to obscure what is really going on, which actually makes it slightly harder to think in Alloy. For example, the on-line tutorial for Alloy describes three distinct ways to think about an Alloy specification [4]:

– "The highest level of abstraction is the OO paradigm. This level is helpful for getting started and for understanding a model at a glance; the syntax is probably familiar to you and usually does what you'd expect it to do.
– "The middle level of abstraction is set theory. You will probably find yourself using this level of understanding the most (although the OO level will still be useful).
– "The lowest level of abstraction is atoms and relations, and corresponds to the true semantics of the language. However, even advanced users rarely think about things this way."

But the page goes on to say, "... the OO approach occasionally [leads to errors]." While we doubt there is a single 'right' way to think, we have found that it helps to think about Alloy specifications in terms of atoms and relations (including the leftmost signature column), as this offers the most direct and accurate mental model of all of the various language constructs.

TLA$^+$ takes a different approach to being accessible to engineers. TLA$^+$ is accompanied by a second, optional language called PlusCal that is similar to a C-style programming language, but much more expressive as it uses TLA$^+$ for expressions and values. PlusCal is automatically translated to TLA$^+$ by a tool, and the translation is direct and easy to understand. PlusCal users do still need to be familiar with TLA$^+$ in order to write rich expressions, and because the model-checker works at the TLA$^+$ level. PlusCal is intended to be a direct replacement for conventional pseudo-code, and we have found it to be effective in that role. Several engineers at Amazon have found that they are more productive in PlusCal than TLA$^+$, particularly when first exposed to the method.

3.7 Flexible and Undogmatic

So far we have investigated three techniques for verifying safety properties using state-based formal methods:

– *Direct:* Write safety properties as invariants, using history variables if necessary. Then check that every behavior of the algorithm satisfies the properties.
– *Refinement:* Write properties in the form of very abstract system designs, with their own variables and actions. These abstract systems are very far from implementable but are designed to be easy to understand and "obviously correct". Then check that the real algorithm implements the abstract design, i.e. that every legal behavior of the real algorithm is a legal behavior of the abstract design.

– *Inductive Invariance:* Find an invariant that implies the desired safety properties. Then check that the invariant is always preserved by every possible atomic step of the algorithm, and that the invariant is also true in all possible initial states.

The first technique (direct) is by far the easiest, and is the only method we have used so far on real systems. We are interested in the second technique (refinement) because we anticipate that such "ladders of abstraction" may be a good way to think about and verify systems that are very complex. We have performed one small experiment with refinement and we intend to pursue this further. We have tried the third technique (inductive invariance), but have found it difficult to apply to real-world systems because we don't yet have good tools to help discover complex inductive invariants.

TLA$^+$ supports the direct method very well. TLA$^+$ also explicitly supports verification of refinement between two levels of abstraction, via a mechanism based on substitution of variables in the high-level spec with expressions using variables from the lower-level spec. We find this mechanism to be both elegant and illuminating, and intend to use it more in the future. (For good examples, see the three levels of abstraction in [24], and the four levels of abstraction in [22].) The TLA$^+$ proof system is designed to use inductive invariants, but the TLA$^+$ model-checker has very limited support for debugging inductive invariants, which is currently a significant handicap.

Alloy supports the direct method fairly well. The Alloy book contains an example of checking refinement [15, p. 227] but the method used in that example appears to require each type of event in the lower-level specification to be analyzed separately, in addition to finding an overall state abstraction function. We have not yet tried to use Alloy to check refinement. Daniel Jackson says [14] that Alloy was designed for inductive analysis, and Pamela Zave used Alloy to perform inductive analysis on the Chord ring membership protocol [32,33]. However, so far we have found inductive analysis of concurrent and distributed algorithms to be very difficult; it currently seems to require too much effort for most algorithms in industry.

Event-B seems to focus on refinement, in the form of derivation. The book [2] and papers [3] on Event-B place strong emphasis on "correctness by construction" via many steps of top-down derivation from specification to algorithm, with each step being verified by incremental proof. While we are interested in exploring verification by refinement (upwards, from algorithm to specification in very few steps), we are skeptical of derivation as a design method. We have never applied this method ourselves, or seen a convincing example of it being applied elsewhere to a system in our problem domain. In general, our correctness properties are both high-level and non-trivial, e.g. sequential consistency, linearizability, serializability. We doubt that engineers can design a competitive industrial system by gradually deriving down from such high-level correctness properties in search of efficient concurrent or fault-tolerant distributed algorithm. Even if it were feasible to derive competitive algorithms from specifications, formal proof takes so much time and effort that we doubt that incremental formal proof can be a viable part of a design method in any industry in which time-to-market is a concern. Our established method of design is to first specify the desired correctness properties and operating environment, and then invent an algorithm using a combination of experience, creativity, intuition, inspiration from engineering literature, and informal reasoning. Once we have a design, we then verify it. For almost all verification tasks we have found that model checking dramatically beats proof, as model checking gives high confidence with reasonable effort. In addition to model checking we occasionally use informal proof, and are keen to try informal hierarchical proof [23]. We do have one or two algorithms that are so critical that they justify verification by formal proof (for which we are investigating the TLA$^+$ proof system). But we doubt that we would use incremental formal proof as a design technique even for those algorithms.

VCC takes a slightly different approach to verification. VCC enables engineers to verify complex global "two-state invariants" (predicates on atomic state transitions) by performing local verification of individual C functions. To achieve this, VCC imposes certain admissibility conditions on the kinds of invariants that may be declared [10]. We don't know if these admissibility conditions may become an inconvenience in practice as our evaluation of VCC was limited to small sequential algorithms and did not verify a global invariant. VCC is based on proof rather than model-checking, but a significant innovation is that the proof tool is guided solely by assertions and "ghost code" added alongside the executable program code. The assertions and ghost code are written in a rich superset of C, so guiding the prover feels like normal programming. VCC seems to support just this one style of verification. We found this style to be engaging and fun when applied to the set of simple problems that we attempted in our evaluation. However, we don't know how well this style works for more complex problems.

4 High Return on Investment

4.1 Handle All Types of Problems

We don't have time to learn multiple methods, so we want a single method that works for many types of problem: lock-free and wait-free concurrent algorithms, conventional concurrent algorithms (using locks, condition variables, semaphores), fault-tolerant distributed systems, and data-modelling. We have used TLA$^+$ successfully for all of these. Alloy is good for data modelling but while preparing this paper we learned that Alloy was not designed for checking concurrent or distributed algorithms [14]. VCC excels at verifying the design and code of low-level concurrent algorithms, but we don't know how to use VCC to verify high-level distributed systems.

4.2 Quick to Learn and Obtain Useful Results on Real Problems

We found that Alloy can give very rapid results for problems of modest complexity. TLA$^+$ is likewise easy to adopt; at Amazon, many engineers at all levels of experience have been able to learn TLA$^+$ from scratch and get useful results in 2 to 3 weeks, in some cases just in their personal time on weekends and evenings, and without help or training. We found that VCC is significantly harder to learn because of relatively sparse documentation, small standard library, and very few examples.

4.3 Improve Time to Market

After using TLA$^+$ to verify a complex fault-tolerant replication algorithm for Dynamo DB, an engineer in Amazon Web Services remarked that, had he known about TLA$^+$

before starting work on Dynamo DB, he would have used it from the start, and this would have avoided a significant amount of time spent manually checking his informal proofs. For this reason we believe that using TLA$^+$ may improve time-to-market for some systems in addition to improving quality. Similarly, Alloy may improve time-to-market in appropriate cases. However, Alloy was not designed to check complex distributed algorithms such as those in Dynamo DB, so in those cases we believe that using Alloy would take much longer than using TLA$^+$, or might be infeasible. VCC is based on proof, and even with VCC's innovations proof still requires more effort than model-checking. We did find that, when developing small sequential algorithms, proof by VCC can take less effort than thorough manual testing of the algorithm. However, we don't know if those results translate to complex concurrent systems.

5 Conclusion

We were impressed with all of the methods that we considered; they demonstrably work very well in various domains.

We found that Alloy is a terrific tool for concurrent or distributed algorithms of modest complexity, despite not being designed for such problems. The analyzer tool has an engaging user interface, the book and tutorials are helpful, and Alloy offers the fastest path to useful results for engineers who are new to formal methods. However, Alloy's limited expressiveness, slightly complicated language, and limited analysis tool, result in a method that is not well suited to large complex systems in our problem domain.

We found Microsoft VCC to be a compelling tool for verifying low-level C programs. VCC does have some features for abstraction but we could not see how to use these features to verify high-level designs for distributed systems. It may be possible, but it seemed difficult and we could not find any relevant examples.

We found that Coq, PVS, and other tools based solely on interactive proof assistants, are too complicated to be practical for our combination of problem domain and time constraints.

Event-B seems a promising method, but we were deterred from actually trying it due to the documentation's strong emphasis on deriving algorithms from specifications by top-down refinement and incremental proof, which we believe to be impractical in our problem domain. However, we later learned that Event-B has a model checker, so may support a more practical development process.

TLA$^+$ is a good fit for our needs, and continues to serve us well. TLA$^+$ is simple to learn, simple to apply, and very flexible. The IDE is quite useable and the model-checker works well. The TLA$^+$ Proof System is a welcome addition that we are still evaluating.

We would welcome further progress in the following areas:

- Model checking: Ability to check significantly larger system-instances.
- Verify the code: Improved tools for checking that executable code meets its high-level specification. We already use conventional tools for static code analysis, but are disappointed that most such tools are shallow or generate a high rate of false-positive errors.

- Proof: Improved support for standard types and operators, e.g. sequences and TLA$^+$'s CHOOSE operator (Hilbert's epsilon). More libraries of lemmas for common idioms, plus examples of how to use them in proofs. Education and tools to help find complex inductive invariants.
- Methods for modelling and analyzing performance: For instance, predicting the distribution of response latency for a given system throughput. We realize that this is an entirely different problem from logical correctness, but in industry performance is almost as important as correctness. We speculate that model-checking might help us to analyze statistical performance properties if we were able to specify the cost distribution of each atomic step.

We would like to thank Leslie Lamport, Daniel Jackson, Stephan Merz, and the anonymous reviewers from the ABZ2014 program committee, for comments that significantly improved this paper.

References

1. Abrial, J.-R.: Formal methods in industry: achievements, problems, future. In: 28th Intl. Conf. Software Engineering (ICSE), Shanghai, China, pp. 761–768. ACM (2006)
2. Abrial, J.-R.: Modeling in Event-B. Cambridge University Press (2010)
3. Abrial, J.-R., et al.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT 12(6), 447–466 (2010)
4. Alloy online tutorial: How to think about an alloy model: 3 levels, `http://alloy.mit.edu/alloy/tutorials/online/sidenote-levels-of-understanding.html`
5. Event-B wiki: Industrial projects, `http://wiki.event-b.org/index.php/Industrial_Projects`
6. Barr, J.: Amazon S3 – the first trillion objects. Amazon Web Services Blog (June 2012), `http://aws.typepad.com/aws/2012/06/amazon-s3-the-first-trillion-objects.html`
7. Barr, J.: Amazon S3 – two trillion objects, 1.1 million requests per second. Amazon Web Services Blog (March 2013), `http://aws.typepad.com/aws/2013/04/amazon-s3-two-trillion-objects-11-million-requests-second.html`
8. Batson, B., Lamport, L.: High-level specifications: Lessons from industry. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2002. LNCS, vol. 2852, pp. 242–261. Springer, Heidelberg (2003)
9. Bolosky, W.J., Douceur, J.R., Howell, J.: The Farsite project: a retrospective. Operating Systems Reviews 41(2), 17–26 (2007)
10. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: Local verification of global invariants in concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 480–494. Springer, Heidelberg (2010)
11. Douceur, J., et al.: Memoir: Formal specs and correctness proof (2011), `http://research.microsoft.com/pubs/144962/memoir-proof.pdf`
12. Hall, A.: Seven myths of formal methods. IEEE Software 7(5), 11–19 (1990)
13. Holzmann, G.: Design and Validation of Computer Protocols. Prentice Hall, New Jersey (1991)
14. Jackson, D.: Personal communication (2014)
15. Jackson, D.: Software Abstractions, revised edition. MIT Press (2012), `http://www.softwareabstractions.org/`

16. Lamport, L.: Comment on the history of the TLC model checker, `http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#yuanyu-model-checking`
17. Lamport, L.: Summary of TLA$^+$, `http://research.microsoft.com/en-us/um/people/lamport/tla/summary.pdf`
18. Lamport, L.: The TLA$^+$ Hyperbook, `http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html`
19. Lamport, L.: The Temporal Logic of Actions. ACM Trans. Prog. Lang. Syst. 16(3), 872–923 (1994)
20. Lamport, L.: Specifying Systems. Addison-Wesley (2002), `http://research.microsoft.com/en-us/um/people/lamport/tla/book-02-08-08.pdf`
21. Lamport, L.: Fast Paxos. Distributed Computing 19(2), 79–103 (2006)
22. Lamport, L.: Byzantizing Paxos by refinement. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 211–224. Springer, Heidelberg (2011)
23. Lamport, L.: How to write a 21st century proof. Fixed Point Theory and Applications (2012)
24. Lamport, L., Merz, S.: Specifying and verifying fault-tolerant systems. In: Langmaack, H., de Roever, W.-P., Vytopil, J. (eds.) FTRTFT 1994 and ProCoS 1994. LNCS, vol. 863, pp. 41–76. Springer, Heidelberg (1994)
25. Lamport, L., Sharma, M., Tuttle, M., Yu, Y.: The wildfire challenge problem (2001), `http://research.microsoft.com/en-us/um/people/lamport/pubs/wildfire-challenge.pdf`
26. Lamport, L., Tuttle, M., Yu, Y.: The wildfire verification challenge problem [example of a specification from industry], `http://research.microsoft.com/en-us/um/people/lamport/tla/wildfire-challenge.html`
27. Leinenbach, D., Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 806–809. Springer, Heidelberg (2009)
28. Lu, T., Merz, S., Weidenbach, C.: Towards verification of the Pastry protocol using TLA$^+$. In: Bruni, R., Dingel, J. (eds.) FMOODS/FORTE 2011. LNCS, vol. 6722, pp. 244–258. Springer, Heidelberg (2011)
29. Newcombe, C.: Debugging designs. Presented at the 14th Intl. Wsh. High-Performance Transaction Systems (2011), `http://hpts.ws/papers/2011/sessions_2011/Debugging.pdf` and associated specifications: `http://hpts.ws/papers/2011/sessions_2011/amazonbundle.tar.gz`
30. Owre, S., et al.: Combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996)
31. Schwartz, B.: The paradox of choice, `http://www.ted.com/talks/barry_schwartz_on_the_paradox_of_choice.html`
32. Zave, P.: Using lightweight modeling to understand Chord. Comp. Comm. Reviews 42(2), 49–57 (2012)
33. Zave, P.: A practical comparison of Alloy and Spin. Formal Aspects of Computing (to appear, 2014), `http://www2.research.att.com/~pamela/compare.pdf`